Comprehensive Characterization of an Open Source Document Search Engine

ZACHARIAS HADJILAMBROU, MARIOS KLEANTHOUS, and GEORGIA ANTONIOU, University of Cyprus
ANTONI PORTERO, IT4Innovations
YIANNAKIS SAZEIDES, University of Cyprus

This work performs a thorough characterization and analysis of the open source Lucene search library. The article describes in detail the architecture, functionality, and micro-architectural behavior of the search engine, and investigates prominent online document search research issues. In particular, we study how intra-server index partitioning affects the response time and throughput, explore the potential use of low power servers for document search, and examine the sources of performance degradation ands the causes of tail latencies. Some of our main conclusions are the following: (a) intra-server index partitioning can reduce tail latencies but with diminishing benefits as incoming query traffic increases, (b) low power servers given enough partitioning can provide same average and tail response times as conventional high performance servers, (c) index search is a CPU-intensive cache-friendly application, and (d) C-states are the main culprits for performance degradation in document search.

CCS Concepts: • **General and reference** → **Performance**; • **Information systems** → **Web search engines**; **Information retrieval**; • **Applied computing** → **Document searching**; • **Hardware**; • **Computer systems organization** → *Architectures*; *Distributed architectures*;

Additional Key Words and Phrases: Document search, index partitioning, parallel index search, parallelism, characterization, real hardware, measurement, evaluation, performance, experimentation

ACM Reference format:
Zacharias Hadjilambrou, Marios Kleanthous, Georgia Antoniou, Antoni Portero, and Yiannakis Sazeides. 2019. Comprehensive Characterization of an Open Source Document Search Engine. *ACM Trans. Archit. Code Optim.* 16, 2, Article 19 (May 2019), 21 pages.
https://doi.org/10.1145/3320346

1 INTRODUCTION

Online search has become one of the most popular and necessary services that virtually all people use for everyday tasks. Online search comes in many forms, such as web search, email search, enterprise search for business records, and landmark search in navigation applications. These applications must respond quickly to user queries. For instance, even slight slowdowns in response times can have negative impact on online services' revenue [3, 12]. Therefore, online

Authors' addresses: Z. Hadjilambrou, M. Kleanthous, G. Antoniou, and Y. Sazeides, University of Cyprus, 1 University Avenue, Aglantzia, 2109, Cyprus; emails: {zhadji01, mklean, ganton12, yanos}@cs.ucy.ac.cy; A. Portero, IT4Innovations, VSB-University of Ostrava, 70833 Ostrava–Poruba, Czech Republic; emails: antoni.portero@gmail.com.

1544-3566/2019/05-ART19
https://doi.org/10.1145/3320346

search services are required to provide tight QoS guarantees, such as tail latencies below 500ms [2] even at peak traffic loads.

Previous work aims at improving the latency, efficiency and cost of operation of search services. In the work of Meisner et al. [27], full system power management is evaluated for a web search workload. To improve energy efficiency, Lo et al. [20] proposed running each server just fast enough to satisfy global latency requirements, whereas Vamanan et al. [33] proposed to exploit time slack by slowing down individual sub-queries. The possibility of using mobile cores for web search for improved cost and energy efficiency is studied in the work of Reddi et al. [30]. Ren et al. [31] examined how web search can benefit from heterogeneous cores, whereas Haque et al. [10] and Jeon et al. [15] looked at adaptive parallelism for improving response times. Work stealing for meeting web search target latency is proposed by Li et al. [17]. Hsu et al. [14] propose a turbo boost framework that increases CPU voltage and frequency at fine-grain time intervals to reduce the latency of computational heavy search queries. Other work has collocated search applications with other types of workloads to increase data center utilization [25, 26, 35].

This article presents a thorough top-down characterization of an open source search engine to improve the overall understanding of search engines. In particular, this work presents a characterization of the Lucene-based Nutch web search benchmark [8] on real hardware providing insights about the application and micro-architectural level behavior of this benchmark. This workload is based on the popular Lucene document search engine. Previous characterization efforts of this benchmark focused only on the query stream characterization [34] and micro-architectural characterization [8]. Another work conducted with the Nutch benchmark [9] evaluated the performance of index intra-server partitioning and slower cores. However, that work used a small index (5GB) and closed-loop query traffic (stress test traffic). This work extends and earlier work [9] by using larger index size (10GB) and Poisson inter-arrival distribution for dictating query arrival rate. Furthermore, this work provides a broader top-down benchmark characterization, as it performs, among other, (a) an analysis of the performance benefit from various micro-architectural features such as caching and prefetching, and (b) an analysis of the the latency impact of DVFS and idle states.

The specific contributions of this work are the following:

(1) We characterize end-to-end query processing times and confirm that index search is the most time-consuming part of the query execution [9, 27, 30].
(2) We show that index search time scales linearly with index size, whereas other operations, such as document summary generation, take constant processing time.
(3) We demonstrate that a 10GB index, generated with a typical crawl walk starting from various seed web sites, exhibits good load balancing in terms of the number of indexed documents per partition.
(4) Given the performance scaling with dataset partitioning and parallel search (due to the good load balancing of index terms across partitions), we motivate the use of low power servers with many simple cores for index search.
(5) We confirm that intra-server index partitioning is beneficial for both average and tail latency reduction, but benefits decrease with increased query arrival rate and load imbalance [9, 10, 15].
(6) The index is stored in compressed form, and each query executes a large amount of instructions for decompression to read its index from memory. This explains the high instructions per cycle (IPC) and the good caching behavior of index search.
(7) Idle states are detrimental to index search performance.

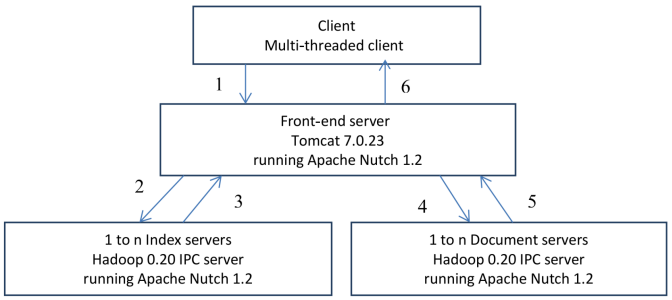

Fig. 1. Basic components of Nutch with arrows showing the query execution flow.

The rest of the article is organized as follows. Section 2 describes the search engine framework used for the experiments in this work. Section 3 presents the experimental setup. Section 4 presents several studies to characterize the application-level performance behavior of the benchmark. Section 5 presents a micro-architectural characterization. The article concludes and provides direction for future work in Section 6.

2 SEARCH ENGINE FRAMEWORK DESCRIPTION

2.1 Architecture and Functionality

This section describes the architecture and the functionality of the Nutch web search engine. The Nutch framework provides a lot of useful functionalities, such as (a) crawling and building the index, (b) partitioning the index among multiple index servers, and (c) coordinating index server execution with a front-end server that forwards the search requests to multiple index servers, collects their answers, and sorts them. Figure 1 shows the components and the overview of the search engine's architecture. The Nutch search engine structure consists of the front-end, index, and document servers, and it is similar to what is described by Barroso et al. [5] as the Google query serving architecture. The Nutch index search is based on the Lucene search engine, which is well-known. For instance, the Twitter's real-time search [6] is built on Lucene. Therefore, this engine is representative of some real-world deployments. It must be noted, however, that compared to the most widely used search engines, such as Google or Bing, the search engine used in this work has some differences mainly in document scoring and query processing, which are discussed later in this section.

In the following, we provide a description of the query-processing flow that is also illustrated in Figure 1. First, the client sends a query to the front-end server. Second, the front-end receives the query and asks each index server to return the most relevant query documents. Third, the index servers perform the search and respond to the front-end with the document IDs and the relevance scores of the top-k relevant matching documents. Fourth, the front-end collects the results and sorts the documents according to their relevance score. In this step, the front-end performs a check for duplicate results. The default configuration of Nutch allows no more than two results from the same site. Depending on the query, steps 2 through 4 may be repeated multiple times until the front-end is satisfied with the number of unique sites. To increase the number of unique sites retrieved, the number of top-k results returned by each index-server is increased by 2X at each search repetition. These subsequent searches are referred to as optimization searches. Fifth, after the front-end has the final top-k results, it sends a detail request to each index server whose search results are in the current top-k list. Each index server responds to a detail request with a title and a URL. Sixth, as soon as the front-end has all the titles and URLs for the top-k results, it asks the document servers for their summaries. The front-end is aware of which documents

Term	Document Frequency	<Document Id, term Frequency>
Architecture	100	<1,1>,<2,3>,<3,1>,<4,100>,<5,1>, …..
Courses	3	<1,2>,<2,1>,<4,1>, …..
CS	4	<1,1>,<2,3>,<5,5>,<9,7>, …..
……	….	……..

Fig. 2. High-level view of the index organization.

each document server holds. Consequently, only the document servers that hold the documents in the top-k results are being asked for summaries. Seventh, the document servers generate the summaries and send them to the front-end. Eighth, when the front-end receives the summaries, it assembles the final HTML response and sends it to the client. Next, we provide a more detailed per-component description.

Index server. The index server is a Hadoop 0.2 IPC server process running the Lucene 3.0.1 search engine. The Hadoop IPC server consists of (a) a listening thread, which listens for incoming requests from the front-end server; (b) the handler threads, which perform the index search or retrieve the details of a document; and (c) a responder thread for sending the responses to the front-end.

Nutch uses document partitioning, which is the most common index partitioning implementation [24]. With document partitioning, each index server holds an index for a disjoint set of documents. The index of our benchmark is generated by a crawling process that uses Hadoop's MapReduce framework [1]. The index is stored in multiple partitions, similarly to a typical Hadoop MapReduce output, with each partition storing a disjoint set of documents. The particular crawler distributes the documents to an index partition using the hash of the document's URL and a modulo operation, such as *partitionNo = URL_hash mod number_of_partitions*. According to theory [24], this document distribution results in a more uniform distribution of query processing time across index servers. This explains the good load balancing we observe later in the experimental results.

Next, we discuss the implementation of the index organization shown in Figure 2. The index terms are stored in an array in alphabetical order. The alphabetical ordering enables binary search and fast term searching. A parallel array holds pointers to a byte stream; each pointer points to the position in the byte stream where the <documentId,termFrequency> pairs of a term start. The list of a term's <documentId,termFrequency> pairs is called the *posting list*. The documentId and termFrequency pairs are compressed using a variable integer format [23]. The variable integer format enables saving space by using the first bit of each byte to show whether more bytes are remaining (if a bit is equal to 1, more bytes are left; otherwise, no bytes are left to read). This way, all numbers from 0 to 127 can be represented with one byte, all numbers from 128 to 16,383 with two bytes, and so forth. The posting list can be sorted by score [15, 24]—for example, the PageRank [28]. Sorting the docs by score is beneficial both for performance and for relevance of results. It provides quick access to the most popular documents that are most likely to fit the user's information needs. In Nutch, all posting lists are sorted by DocId, which is useful for performing efficient merging of posting lists for conjunctive (AND) queries (posting list intersection) [24].

Now we discuss the index search procedure. Nutch uses conjunctive multi-term queries, a vector space model (for representing documents), and a tf.idf weighting scheme (for ranking documents) [22]. The actual search procedure goes like this. First, binary search is performed to find the posting lists for all query terms. Then, the posting list intersection is performed to find the documents that contain all query terms. For each document found, a tf.idf score is calculated. The tf.idf [22] weighting scheme gives high scores to documents that have many occurrences of the query terms and also to documents that contain many occurrences of rare terms. To decide the top-k (e.g.,

top 10) most relevant to the query documents, the index server sorts the documents based on their tf.idf score. To summarize, the index search time for a conjunctive query is determined by (a) the binary search for finding each term's posting list, (b) the merging of the term posting lists, (c) the ranking of the documents that come out of the intersection, and (d) the sorting of the documents by their relevance rank.

At this point, it is useful to provide a brief comparison of Lucene with other search engines. The posting list intersection is generally considered the most time consuming part of the index search procedure, not only for Lucene but for other search engines as well [32]. In terms of the ranking function, however, despite tf.idf being a very well known scoring scheme, some differences are to be expected as compared to the most advanced search engines. For instance, the Bing search engine uses machine learning–based ranking [30]. In addition, web search engines usually perform an early termination of the search procedure either by using a cut-off latency [30] or when the quality of results is unlikely to improve with further searching [15]. Early termination is used to avoid having an index server search for too long. Nutch provides an option to stop searching according to a cut-off latency. With a cut-off latency search configuration, queries that are prematurely stopped may suffer from poor result quality (low relevance of the search results). For instance, this may happen when an index server is, for a variety of reasons, slowed down and does not have enough time to go through its posting list. In practical terms, it is not trivial to compare server performance in terms of relevance of query results. Given that optimizations that expedite search time can be beneficial to deployments with and without cut-off, for our evaluation we use the default configuration of the Nutch benchmark, which does not use any cut-off latency. Consequently, the various server configurations that we assess are compared using latency metrics (both average and percentile).

Two other parameters that are important for the index search are (a) the amount of index dataset an index server holds and (b) the number of index servers used. The larger the dataset an index server holds, the longer its search time. By increasing the number of index servers, we can reduce response time. Using more index servers enables increasing the degree of partitioning, which means that each index server gets a smaller index part and thus needs less time to respond to a query. Of course, it is important to have balanced posting lists across the partitions; otherwise, partitioning will not provide the expected performance benefit. Index partitioning can be done across servers or inside the same server (intra-server partitioning) [15]. Intra-server partitioning is realized by running multiple index search contexts on the same server with each context working on a different index part. Intra-server partitioning represents a trade-off between throughput and response time latency. Having many index searchers in a CPU socket can speed up the execution of a query but reduces the number of available cores for handling multiple queries in parallel and can thus increase queueing time.

Document server. The document server, like the index server, is an instance of a Hadoop IPC server with a listening thread, a responder thread, and many handler threads. The document server contains the actual copies of the documents. The document server is used for fetching the summaries of the documents. Summaries can be dynamic or static [24]. Static summaries are preloaded and are always the same regardless of the query that hit a document, whereas dynamic summaries are query dependent and attempt to explain to the user why a document is relevant to the query. Nutch uses a dynamic summarizer. The dataset of the document server, like the index, is partitioned in many parts. Each part contains a disjoint set of documents. For expediting access to the document's content, the document server uses a partitioning function (which takes as input a document's URL) for determining at which partition the document is located. A document server partition contains (URL, web page content) key value pairs and a small index that points to a fraction of keys that further expedites the summary generation procedure.

Front-end server. The front-end is a Tomcat web server running the Nutch application. Tomcat is multi-threaded and spawns a new thread for handling a new query request. The front-end coordinates the entire query execution and is the component that acts as a link between the client and the nodes that do the actual job: the index and the document servers.

Client. The client is a process thread used for sending queries to the front-end. A client thread can send queries based on some inter-arrival time distribution [15] (open-loop) or in a stress test manner (closed loop). In the stress test scenario, a client thread sends a new query as soon as it receives the response for the previous query sent.

Having discussed the architecture and the information flow of the search engine, we next describe its inputs, such as the queries.

2.2 Inputs

Search engines can handle various types of queries. The most typical type of queries are multi-term queries. For a multi-term query, search engines try to give a higher score to a document that contains many terms of the query (e.g., 3 of 5) [24]. Some search engines allow multi-term queries to be combined with Boolean expressions like OR, AND, NOT. Other types of queries are (a) phrase queries, which try to find documents that contain a phrase specified by a user, and (b) proximity queries for terms within a specified distance [24]. Phrase queries and proximity queries can be implemented with a positional index that holds the position of each term in a document. Nutch considers all multi-term queries as AND queries, and it can also execute phrase queries. For testing a search engine, we can either use a real traffic query stream trace or random queries built using index terms.

2.3 Metrics

A search engine must provide both low mean and low high percentile response times. Service guarantees, such as 99% of response times within 500ms, are usually set to keep users satisfied [2]. Such service guarantees must be preserved even at the highest (peak) loads [7, 16]. Relevance of the search results is also a crucial factor that contributes to user satisfaction and the eventual success of a search engine. The relevance of search results can be improved with more sophisticated ranking functions and a larger index [15].

2.4 Sources of Performance Degradation, Variability, and Tail Latencies

Search engines in real-world deployments, such as web search running in data centers, perform index search across thousands of servers in parallel. At such grand scale, performance variability is more likely to happen. Subject to a particular setup, a single node can either slow down the whole query execution or may negatively affect the relevance of results (in the case of utilizing cut-off latency) [4]. In any case, the user experience is affected negatively. Search engines suffer from intrinsic performance variability, as some queries require longer time to execute than the average case. These queries plus the various sources of performance variability [18], such as power-saving features, are the main culprits for high tail latencies. Part of this work aims to identify and address sources of performance variability and of poor response times in search engines. In particular, in Section 4.6, we analyze the performance degradation caused by power-saving features such as idles states and dynamic voltage frequency scaling (DVFS).

3 EXPERIMENTAL DETAILS

For the experimental analysis, we use dual-socket blade servers based on an Intel Xeon E5-2665 @2.4GHz CPU. The processor supports frequency scaling with a range from 1.6 to 2.4 GHz. Table 1 provides details about the blade server hardware. At most, we use four blade servers, one for

Table 1. Server System Parameters

Number of CPUs (sockets)	2
CPU	Intel Xeon E5-2665 @ 2.4GHz
Micro-architecture	Sandy Bridge
Private L1 (Instruction + Data)	32KB+32KB
Private L2	256KB
Shared L3 size	20MB
Number of physical cores per CPU	8
Number of logical cores (SMT contexts) per CPU	2
DRAM	8 x 8 GB DDR3 1,600MHz
NUMA Memory nodes	Two with each node having 32GB
Ethernet speed	InfiniBand
OS	Ubuntu 16, Kernel 4.4.0-63

each of the following functions: client, front-end server, index server, and document server. The experiments that focus on index search are conducted with three blade servers (client, front-end, index). The machines are connected with an InfiniBand communication network.

Now we explain in more detail how we performed the experiments. The client machine runs one client process that we implemented. For almost all experiments, the client sends queries using a Poisson distribution to determine inter-arrival times. The client can also send queries in a stress test manner. We use the stress test with one client thread only for the experiments in Sections 4.1 and 4.2 to isolate query stream characterization and processing time breakdown from queueing delays and resource contention overheads. The front-end machine runs a Tomcat web server process that spawns a new thread for handling each new query request.

Unless stated otherwise, our experiments use the following default configuration. The index and document machines run one index server process and one document server process, respectively. We keep the number of handler threads (see Section 2.1) of the index and document servers equal to the number of cores. We pin the handler threads to specific cores and memory nodes using the numactl command [19], and we disable DVFS (run constantly at the same nominal frequency). In addition, in all runs, we place the index in a memory-mapped file system using the tmpfs Linux command. Explicitly pre-loading the index into memory, instead of relying on the OS file system cache, improves performance and reduces variability between runs. Moreover, the servers are run with hyper-threading disabled to ease the analysis, as this avoids performance differences arising from running in two different physical cores versus running on two logical cores mapped in the same physical core. Again for analysis ease, turbo boost is disabled because it can cause variation in measurements. Finally, we disabled all C-states because we find that they cause a significant increase in latencies (Section 4.6).

Regarding the query stream, we use 100K queries from the AOL query log [29]. We have selected 100K unique queries. This means that each query appears only once in the query stream. This is done to emulate the effect of common or repeated queries being captured by a query cache. We are aware of the controversy surrounding the AOL query log because of privacy issues. We want to state that we do not use the AOL log for identification of people. We use the real-life representative queries only for performance characterization and analysis purposes.

For the index dataset, we use a 10GB index crawled from various Internet pages using the Nutch crawler. To crawl and generate the index, we followed instructions in the Nutch tutorial [1]. The dataset is divided into chunks of approximately 220MB each. This facilitates index partitioning since these chunks can be combined to form larger index partitions. Besides the index dataset, the

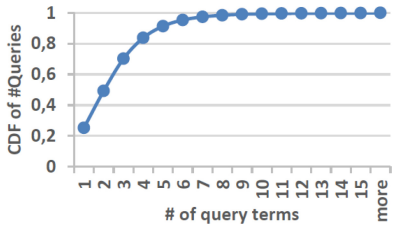

Fig. 3. Frequency of query lengths of the input query stream.

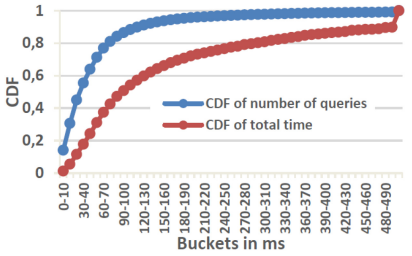

Fig. 4. CDF of the response times. Mean is 59ms, 90th is 118ms, and 99th is 437ms.

crawling also produced a 100GB segment dataset that is given as input to the document server (segment dataset).

In Section 5, we show micro-architectural data such as IPC, captured using the perf Linux utility.

4 CHARACTERIZATION AND ANALYSIS

4.1 Query Stream

The analysis in this section is obtained with one client thread running in stress mode. The client sends the next query only after it receives the answer for the previous query. This is done to isolate the response times from the effects of queuing and resource contention when serving multiple queries in parallel. This way, the analysis is focused only on the time needed to process queries.

Figure 3 shows the cumulative frequency of the queries according to their number of terms. We have observed query lengths from 1 term to 72 terms. The short queries dominate the query stream. On average, the query length is 2.93 terms. A cumulative distribution of the frequency of response times is shown in Figure 4. The average response time is 59ms, the 90th percentile is 118ms, and the 99th percentile is 437ms. Most queries are executed fast, but there are few queries that require hundreds of milliseconds. Figure 4 also shows the same distribution but weighted with the response time of each query. A noticeable fraction of the total processing time, around 10%, is due to queries with very long response times (more than 490ms).

Figure 5 presents the correlation between the number of query terms and the average end-to-end response time. The error bars represent the observed maximum and minimum for a given number of terms. The figure also shows, on the secondary y-axis, the average number of documents matched for a given number of query terms. Please note that both y-axes are logarithmic. The results in Figure 5 clearly show an increase in the average response time with a growing number of query terms. This behavior is consistent with what previous work has observed [32]. The correlation with matching documents is a rather monotonic decrease with an increasing number of query terms. We observe a high deviation in response times that decreases as the number of query terms grows. In the following, we discuss the implications of these observations.

According to Tatikonda et al. [32], the intersection of posting lists dominates the processing cost. This explains why, on average, we observe higher response times for larger queries. The large deviation in response times, especially for queries with a small number of terms (e.g., 1 to 3 terms), can be explained by considering the number of documents that are relevant to a query. Let us take, for example, some single term queries. There are single term queries like “about” or “home,” which are virtually relevant to all documents in the dataset. The processing cost of ranking all of those documents plus the possible extra optimization searches, which may be required for duplicate elimination, yields high response times. The wide deviation in response times for queries of the same length underlines the difficulty in developing accurate heuristics for predicting response

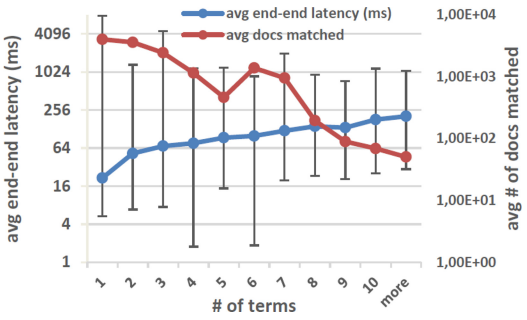

Fig. 5. Average response time and average number of matching documents as a function of the number of query terms.

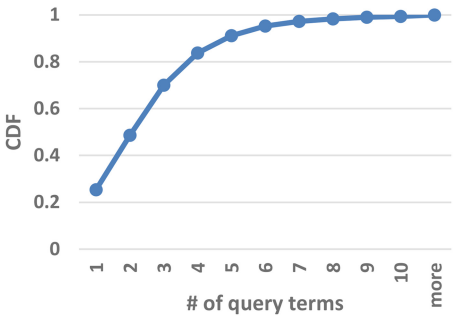

Fig. 6. The breakdown of 99th tail latencies as a function of the number of query terms.

time based only on the number of terms. The reason the documents matched are decreasing with increasing query terms is due to conjunctive multi-term queries. For these queries, as the number of terms grows, it gets more difficult to find documents that contain all the terms of a query.

The results in Figure 6 show the breakdown for another key search engine characteristic: the contribution of queries with a given number of terms to long tail latency (>99%). The results in this figure have a correlation with Figure 3 that shows the breakdown of the queries according to their number of terms. The large contribution of queries with few terms to tail latency is due to their high frequency in the entire stream, as shown in Figure 3. In addition, as we have already mentioned, queries with few terms or even one term, such as "about," may end up with many optimization searches. This kind of queries are very time consuming, as they need many searches and have a lot of matching documents. The bottom line of this analysis is that we do not observe the tail being dominated only by queries with many terms. Small term queries also experience long latency. Similar behavior is observed by Hsu et al. [14].

4.2 Performance in Relation to Dataset Size

Next, we report on the findings about the sensitivity of the response time to the dataset size. The goal is to identify how much processing time is spent in the various phases of the benchmark as a function of the dataset size. According to Section 2.1, we summarize and report the query processing time in terms of the latency of the following four discrete Nutch phases: (a) the time spent for client– front-end communication (the client sends a request, and the front-end assembles the html response and sends it back to client); (b) the index search, which is performed on the index server (including any optimization searches for duplicate deletion, any time the front-end spends to sort the results, and the time for network communication between the front-end and index servers); (c) the detail requests, which are also performed on the index server; and (d) the summary requests, which are performed on the document server. Like in the previous section, we use one client in stress mode to isolate the processing time from queuing delays and resource contention overheads. We perform six experiments with increasing dataset parts both for the index and the document servers.

The result of the experiments is shown in Figure 7. It can be observed that the processing time for the search increases linearly with the dataset size, whereas the rest of the processing times remain mostly the same. It is interesting to understand why the summary generation time does not increase with the dataset size. The explanation lies in the functionality of the document server, described in Section 2.1. The document server uses a hash function to determine at which one of

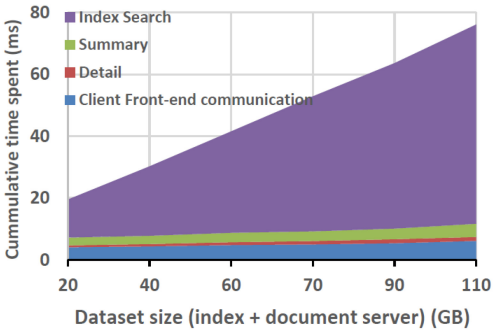

Fig. 7. Cumulative time spent at each phase of the benchmark as a function of dataset size.

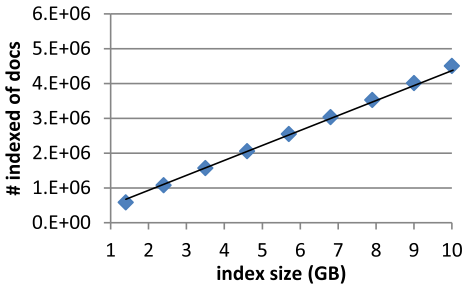

Fig. 8. Numbers of documents stored in an index versus the index size in gigabytes.

the partitions a document is located, so it does not have to search all partitions to find a document. This behavior helps limit the time required for finding a document to be, more or less, the time needed for searching the average size of a document server segment partition, which does not vary a lot across runs, plus the time required for dynamically generating a summary for a document. This targeted search approach cannot be implemented for index search, as a relevant document may exist in any index dataset partition, and thus the search time increases as we increase the index dataset. Increasing the index dataset results in the adding of more documents. This is confirmed by the strong correlation seen in Figure 8 between the number of indexed document increases and the index size. This means that with a bigger index dataset, the index server needs more time to traverse a term's posting list. This explains the nearly linear relation between the index search time and the index size. Such behavior suggests that index search may benefit from index partitioning and parallel search. We explore this possibility in Section 4.4.

In addition, it is clearly shown in Figure 7 that the index search dominates the end-to-end response time. This observation confirms what previous work [15, 27, 30] observes, namely that index search is the most processing-demanding part of a search engine. Hence, for the rest of the article, we perform experiments that focus only on index search and do not perform a query's detail and summary operations. Thus, for the remaining experiments, we use three blade servers: one server running the client, one server running the front-end, and one server running the index server (or index servers for the runs with intra-server partitioning).

4.3 Performance as a Function of Inter-Arrival Rate

In this section, we perform an investigation into how the query arrival rate affects the response latencies and CPU utilization. Figure 9 shows the average latency and Figure 10 the 99th percentile latency, as well as the index server CPU utilization (right axis) for various query arrival rates. Both average and 99th percentile latency increase with higher arrival rates. In particular, average latency increases from 40ms at 50 queries/sec to 125ms at 225 queries/sec, and correspondingly, 99th percentile latency increases from 255 to 1152 ms. The latency increase is the result of (a) longer queue latency and (b) increased contention on shared resources when multiple cores are engaged simultaneously. Higher traffic rates increase both the number of cores that simultaneously must be active, as well as the time queries are queued waiting to be served. Hence, higher arrival rates result in higher latency. Regarding index server CPU utilization, it increases linearly with arrival rate. We stop the exploration at 225 queries/sec, which causes a CPU utilization of 83%. The reason we stop at 225 queries/sec is because at this rate, we have observed some dropped queries due to

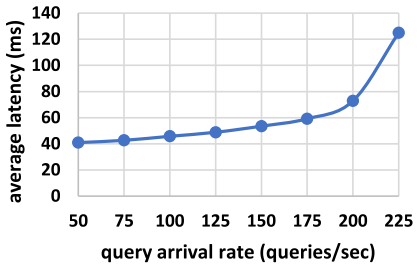

Fig. 9. Average latency for various query arrival traffic rates.

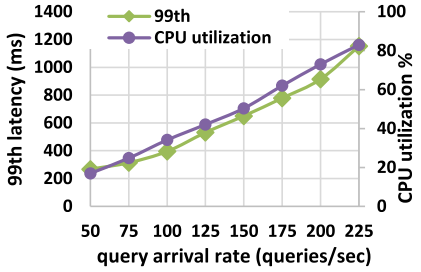

Fig. 10. CPU utilization and 99th percentile latency for various query arrival traffic rates.

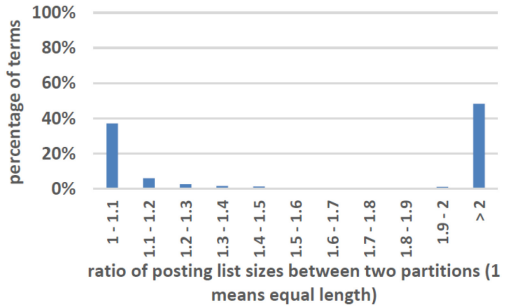

Fig. 11. Histogram of the ratio of posting lists size of all terms for two equal-size index partitions.

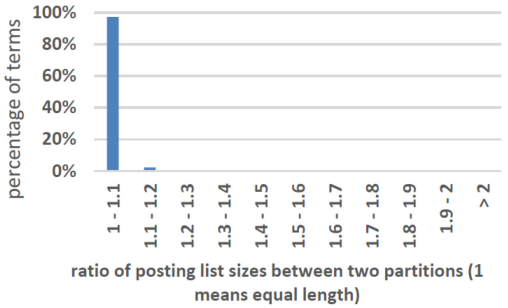

Fig. 12. Histogram of the ratio of posting lists of the most popular terms (at least 1,000 postings) for two equal-size index partitions.

extremely high queuing times. For the rest of the article, to avoid runs with dropped queries, we perform experiments with arrival rates ranging between 50 and 200 queries/sec.

4.4 Intra-Server Partitioning

We now compare the posting lists of two equal-size (in gigabyte) partitions. If the two partitions have equal-size posting lists for most terms, then index partitioning and parallel search can help reduce index search latency. Figure 11 shows a histogram of how balanced the two partitions are according to the size of the posting list for each term. The balance is measured in terms of the size ratio of the biggest over the smallest posting list. Values close to 1 mean equal-size partitions. The results show that the majority of terms have a ratio of above 2. Keep in mind, however, that many terms have very few postings. A query term with a short posting list has negligible contribution to the query's computation time. Therefore, it is more useful to check the load balancing for terms with large posting list sizes (e.g., at least 1,000 documents). The histogram in Figure 12 shows the load balance across partitions for these more popular terms. The figure clearly shows a high degree of load balancing, with 97% of terms having posting list size ratios below 1.1. Consequently, we can conclude that there is a good load balance among the popular terms, which are the ones that add considerable computation time to the query execution. These findings indicate that it is worth examining further index partitioning. Next, we investigate the best server configuration to run a partitioned index and the benefits from a partitioned index search in a realistic setup.

A machine running an index server can have multiple index server processes with each process working on a separate index part. Using many index server processes can reduce the response

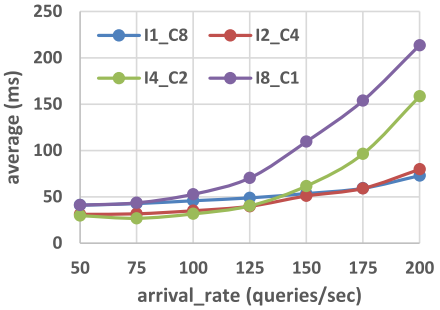

Fig. 13. Average latency for the various index-partitioned configurations across different query arrival rates.

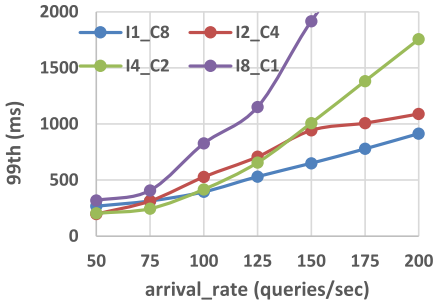

Fig. 14. The 99th percentile latency for the various index-partitioned configurations across different query arrival rates.

latency, due to the good load balancing among index partitions, but at the expense of executing less independent queries in parallel. We study four different configurations for how to run the index server on a CPU. In particular, a socket in our Intel Xeon E5-2665 has eight cores, and we explore the following options: (a) one index server process with eight cores, (b) two index server processes and four cores per index server process, (c) four index server processes and two cores per index server process, and (d) eight index server processes with one core per index server process. We used the numactl command [19] to pin processes to specific cores. To test the sensitivity of each option to incoming traffic, we used arrival rates from 50 to 200 queries/sec. For presentation clarity, we will denote the different configurations with the coding IX_CY, where X denotes the index servers and Y the cores per server (e.g., I2_C2 means two index server processes and two cores per index server process). The results of this analysis are reported in Figure 13 (average latency) and Figure 14 (99th percentile).

We observe that eight index servers (I8_C1) is a bad choice because even at the lowest arrival rates, it does not manage to provide faster latencies than the no-partitioned configuration (I1_C8). This underlines the diminishing benefits from excessive parallelism. However, we observe that the two other partitioned configurations, I2_C4 and I4_C2, behave much better. I4_C2 provides faster average latency than I1_C8 for arrival rates up to 125 queries/sec. I2_C4 provides faster or equal to I1_C8 average latency for all query arrival rates. But both I2_C4 and I4_C2 fall behind I1_C8 in 99th percentile latency for traffic rates higher that 100 queries/sec. In general, we observe that at low utilization levels (below 125 queries/sec), the partitioned configurations I2_C4 and I4_C2 provide faster or equal to I1_C8 average and 99th percentile latency. This is expected because, as pointed out in previous work [15], partitioning is beneficial during periods of low utilization. During low utilization, there is minimal queuing delay, and hence the partitioned setup only gets benefits from parallel query processing that result in lower latency as compared to the non-partitioned setup. At higher traffic rates (above 100 queries/sec), the partitioning benefits are offset from the queuing delays, and hence the configuration without partitioning (I1_C8) provides the best 99th percentile.

The intra-server partitioning can be detrimental at some traffic rates because the speedup from partitioning is not ideal. For example, a two-way partitioning does not provide 2X faster latencies but perhaps 1.9X. This is expected because of imperfect load balancing and parallelism overheads. In the ideal scenario, where partitioning speeds up a query by the degree of partitioning used, the latencies cannot get worse than no partitioning. This is the case because the benefits from reducing an individual query's processing time is at worst offset by an increase in its queuing time, waiting for the other queries to complete (that with no partitioning are processed in parallel). But of course

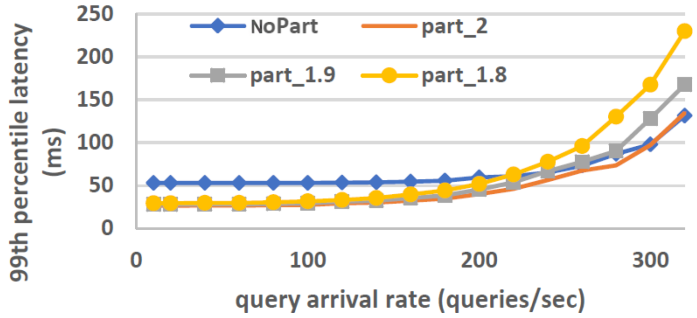

Fig. 15. The 99th latency calculated using a synthetic queuing simulator.

this is not what we observe in our results; what we observe is that after a particular arrival rate, the queuing increase overshadows the partitioning speedup.

We attempt to generalize our intra-server partitioning observations with a synthetic queueing simulator. The queuing simulator accepts the following as inputs: (a) Poisson-distributed query inter-arrival times and the mean of the distribution, (b) a sequence of query processing times obtained from real hardware measurements, and (c) the number of cores for serving independent queries. We run the simulator with a partitioned setup that uses two cores per index server and a no-partitioned setup with four cores per index server (essentially, we mimic an index configuration of I2_C2 and I1_C4). The no-partitioned processing times are the index search times captured on real hardware with a no-partitioned configuration. For the partitioned setups, the search times are equal to the no-partitioned index search times divided by various assumed speedup factors (the ideal speedup is equal to 2).

Figure 15 shows the 99th percentile latency as a function of the arrival rate ranging from 10 queries per second to 320 queries per second, and for various configurations, namely no partitioning and partitioning with speedup of 1.8, 1.9, and 2. The graph shows that when we have speedup of 2×, partitioning performance is always better or the same with the no-partitioned setup. This is what we expected; at low utilization, the ideal partitioning would be a clear winner, but at high utilization, the speedup is nullified by queueing, which results in identical latencies between partitioned and no-partitioned configurations. In general, we observe that the partitioning benefits are decreasing with increasing arrival rate. Under high traffic, the queuing delay offsets the benefit from partitioning, and for configurations with non-ideal speedup, there is a crossover inter-arrival rate after which partitioning gets worse than the no-partitioned configuration. Moreover, the lower the speedup from partitioning, the more shifted to the left is the query arrival traffic rate at which we observe the crossover point.

4.5 Index Server Performance in Relation to CPU Frequency

In Section 4.2, we showed that index search time grows linearly with index dataset size. This behavior provides good motivation for exploring the use of low power servers for document search with index dataset partitioning. Previous work has investigated the use of low power servers for web search [21, 30] but without considering the implications of partitioning.

To approximate the performance of a low power server, we reduce the frequency of our Intel Xeon CPU from 2.4 to 1.6 GHz (the lowest allowed frequency for this CPU). Figure 16 shows that for 1.5x less dataset, the Xeon@1.6GHz achieves the same average response time as the Xeon@2.4GHz. This may indicate that, for the Lucene search engine, with sufficient degree of partitioning, using more low power servers can match the performance of high performance servers.

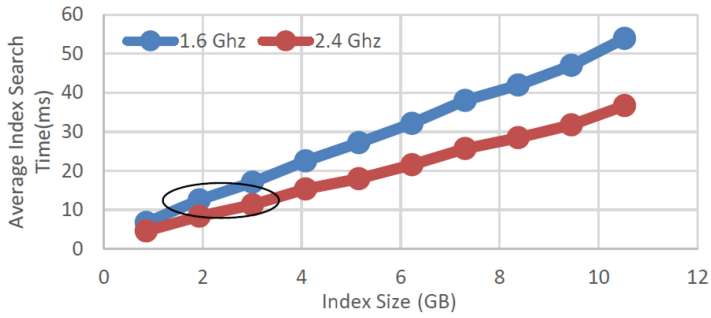

Fig. 16. Index search time for an index server running at 2.4 and 1.6 GHz versus index dataset size. The 1.6GHz achieves the same response times for 1.5x less dataset (e.g., the points in the oval shape).

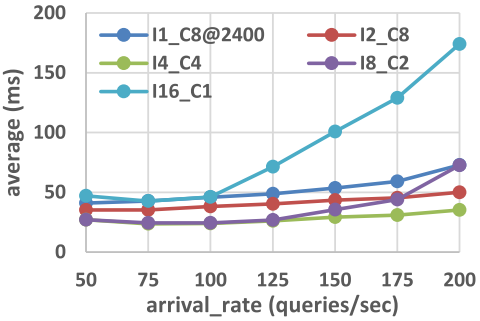

Fig. 17. Average latency for various 1,600MHz index-partitioned configurations across different query arrival rates versus the I1_C8@2,400MHz configuration.

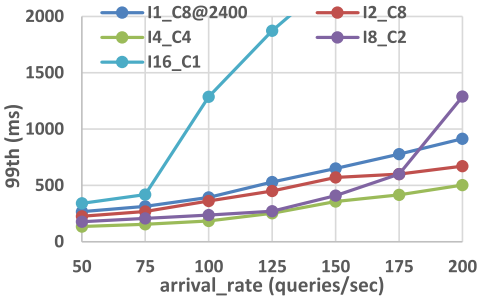

Fig. 18. The 99th percentile latency for various 1,600MHz index-partitioned configurations across different query arrival rates versus the I1_C8@2,400MHz configuration.

Figures 17 and 18 show how two sockets of 1.6GHz Xeons compare to one socket of a 2.4GHz Xeon in terms of average and 99th percentile, respectively. We select as baseline configuration the I1_C8@2.4GHz, which achieves the best 99th percentile latency at high traffic rates (according to Figure 13). Again, we observe that excessive parallelism does not provide good latencies, with I16_C1 being the worst configuration in terms of 99th percentile for all arrival rates. But all the rest of 1,600MHz-partitioned configurations provide lower average and lower 99th percentile as compared to the baseline for all query arrival rates (with the exception of I8_C2 at 200 queries/sec). From this analysis, we conclude that many low power servers can help improve throughput, as stated in previous work, but also response time when enough partitioning is applied.

Arguably, our analysis gives some advantage to the 1.6GHz setup as we assign to each slow server 2x less dataset instead of 1.5x less (the 1.6GHz is only 1.5x slower than 2.4GHz). But this may be desirable for another reason. When using a larger number of low power servers, as compared to high power servers, it becomes more likely that a server will slow down, due to poor load balancing, the whole query execution [5, 13]. Giving each low power server less amount of work reduces the chance of a single server slowing the whole query execution. However, performance is not the only metric that concerns the design of a data center operator; a total cost of ownership (TCO) comparison must be done to justify the benefits of a search engine deployment built with more low power servers as compared to fewer high power servers.

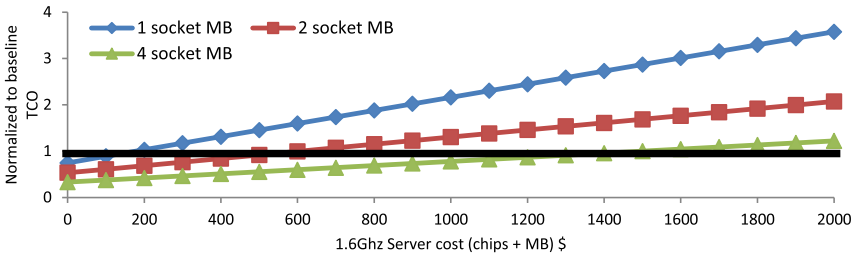

Fig. 19. Relative to baseline (2.4GHz data center) TCO of a 1.6GHz server data center. We see that the more sockets per MtB, the more money the 1.6GHz can cost without surpassing the TCO of the 2.4GHz.

An analysis is performed to estimate how much the cost should be for the 1.6GHz server (motherboard (MtB) plus chips acquisition cost) to be profitable in terms of TCO in comparison to the baseline. Our baseline is a data center of single-sockets servers with E5-2665 CPUs running at 2.4GHz. The low power servers must offer high density and high integration to be profitable. We study three integration scenarios: (a) a pessimistic, single-socket MtB, (b) the scenario we actually used with a dual-socket MtB, and (c) an optimistic scenario with a four-socket MtB. A data center using 1.6GHz processors has twice the processors of the baseline 2.4GHz configuration. The amount of disks, DRAMs, and average data center utilization remains the same. We assumed that with lower frequency and voltage, the 1.6GHz power consumption falls to 34W peak and 13W idle as compared to 80W peak and 30W idle at 2.4GHz. We estimated the baseline data center's server cost to be $616 assuming that a single-socket MtB costs $225 plus the price of a $391 processor.

We estimated TCO using the tool by Hardy et al. [11]. The TCO results, shown in Figure 19, reveal that with more integration, the low power servers can be more profitable. With a single-socket MtB and server cost up to $200, the 1.6GHz option is more profitable. For a dual-socket cost of up to $600 and for a four-socket cost up to $1,500, it is more profitable to use 1.6GHz instead of 2.4GHz processors.

4.6 Performance Sensitivity to DVFS and C-States

In this section, we analyze how idle states and DVFS affect the index search performance. Enabling C-states and DVFS is expected to be detrimental for latency because of the time overhead to transition between idle to active state and to transition from low- to high-voltage frequency levels. We perform this evaluation using the 1I_8C@2.4GHz configuration with 150 queries/sec incoming traffic. Figure 20 shows how the latencies are affected when enabling/disabling C-states and DVFS. The first configuration, cstate(OFF)_DVFS(OFF), has both C-states and DVFS disabled, and this is how we run all the paper experiments. The second configuration keeps C-states disabled but enables DVFS. The latencies are insensitive to the change. The third configuration keeps DVFS disabled but enables C-states. This increases the average latency by 1.5X and the 99th percentile latency by 1.35X as compared to the baseline configuration, cstate(OFF)_DVFS(OFF). It seems that the core wake-up latency significantly increases latency. Disabling DVFS does not improve the performance, as the DVFS governor appears to be performance aware and keeps the cores at the highest voltage-frequency level, at least for the inter-arrival rate used in this experiment.

5 MICRO-ARCHITECTURAL CHARACTERIZATION AND OPTIMIZATIONS

This section presents a micro-architectural characterization of the index search and confirms from a micro-architectural point of view the application-level insights of Section 4. In addition, this

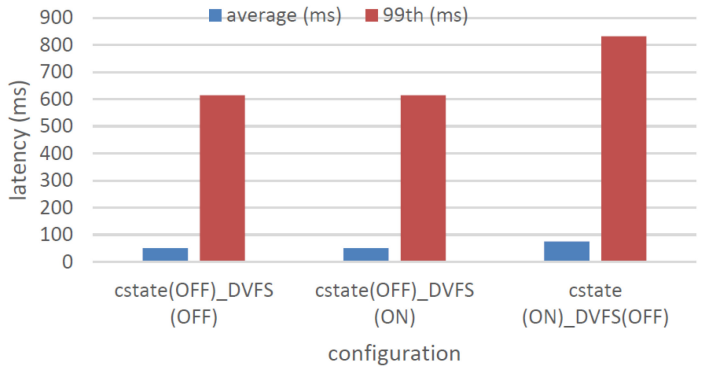

Fig. 20. Effect of DVFS and C-states on latencies.

```
byte = data[i++];
 int value = byte & 01111111; //get the first byte value
for ( shift = 7; (byte & 10000000) != 0; shift += 7) //while first bit not equal to zero
{
        byte = data[i++];  //read next byte
        value |= (byte & 01111111) << shift; //get the value of next byte, add it to
        value
}
```

Fig. 21. The decompression algorithm. We extracted this algorithm from the Lucene source code.

section discusses how index search micro-architectural characteristics can be leveraged to improve the efficiency of document search engines.

By utilizing performance counters, we have estimated that, on average, the IPC of an index search operation is 1.5 and the last-level cache (LLC) misses per thousand instructions (MPKI) is 0.77. These numbers imply that index search is a fairly compute intensive application with cache-friendly behavior. Index search has low L3MPKI (L3 is the LLC in our system) despite the fact that the index dataset size is equal to several gigabytes. One explanation for this behavior is that for each query, the index server does not read a lot of data from the memory. We estimate (by taking into account the posting lists of each query's terms) that for the 100K AOL query stream we use, the average per query amount of postings that the index server must read is approximately 100,000 postings. Assuming 8-byte postings (4byteDocid, 4byteTermFreq), the amount of postings read is equal to 800KB, which is not much. Moreover, the postings are compressed (see Section 2.1). The compression reduces further the postings' size and consequently the amount of bytes read during index search.

The other reason for the high IPC and the low L3MPKI is the compute-intensive nature of the postings decompression algorithm. During index search, the index server must decompress all the terms' postings related to a query. Decompression takes a significant portion of the index search execution time, and we have measured that, on average, 37% of the index search time is spent on decompression. To estimate the time spent on decompression, we instrumented the source code with the Java's nanoTime function. The bottom line is that a significant portion of index search is spent on decompression, and this means that the decompression algorithm can significantly affect the overall index search operation's IPC and cache behavior.

Next, we explain why decompression is compute intensive. To decompress the postings, we have to read the byte stream byte by byte and perform the steps shown in Figure 21. We can identify at

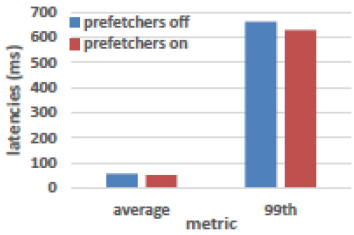

Fig. 22. Impact of disabling prefetching on latencies.

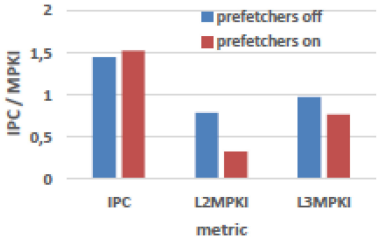

Fig. 23. Impact of disabling prefetching on IPC (higher is better), L2MPKI (lower is better), and L3MPKI (lower is better).

least 8 instructions per byte read. If we multiply that by 64 bytes, which is the size of a cache block, then we get 512 instructions executed per cache block fetched into the LLC cache. Assuming a cold cache, this would translate approximately to a L3MPKI of 2. The actual index server's L3MPKI is slightly lower for various reasons. Prefetching or postings cached from previous queries can reduce the L3 misses. In addition, the actual low-level code executes more than the 8 instructions identified in Figure 21 per byte read. For example, the actual code also increments a variable that counts how many postings of a term are read and checks if the document frequency number is reached. Our findings are similar to what previous work has reported. In particular, Barroso et al. [5] show that the Google Search index server exhibits good caching behavior and low memory bandwidth utilization due to spatial locality in index data accesses and a fair amount of computation required for every block fetched into cache.

The compute-intensive nature of index search and the small memory pressure support the following observations made in Section 4. First, in Section 4.5, the performance ratio between the 2.4G and 1.6 GHz cores is exactly 1.5 (equal to the ratio of frequencies); this is a result of the fact that the memory time during index search is small. Second, in Sections 4.4 and 4.5, intra-server partitioning is not negatively affected by using multiple index server processes; if index search were a memory-intensive process, then running in parallel of multiple search processes would result in increased contention due to memory accesses.

The small memory pressure makes index search rather insensitive to cache optimizations. This insensitivity may open opportunities for gains in other metrics, such as power consumption, data center utilization, and cost. For example, index search could probably be run on processors with a smaller LLC cache; those chips can be cheaper and less power hungry [21]. Next, we study how prefetching, query temporal locality, and cache size affect the index search performance. We show that these cache optimizations have a rather small impact on index search times.

We begin with the prefetching. We run our baseline configuration (1I_8C@2.4GHz and 150 queries/sec traffic) with all hardware prefetching disabled to evaluate how prefetchers affect the latencies and the micro-architectural behavior. Figure 22 shows the average and 99th percentile latency while Figure 23 shows the IPC, L2MPKI, and L3MPKI for the baseline configuration with prefetchers enabled (the default way we run in the rest of the article) and with prefetchers disabled. As expected, we observe an increase in L2MPKI and L3MPKI. In particular, the L2MPKI increase nearly 2.5X times with prefetching disabled. However, the impact on IPC and latency is rather small; we observe a 5% increase in IPC and 99th percentile latency. This implies that prefetching improves a small portion of the query execution time. The time spent for reading the posting lists from memory is short compared to the time spent on processing the postings. Thus, even though prefetching decreases the cache misses, the impact in total index search time execution is small.

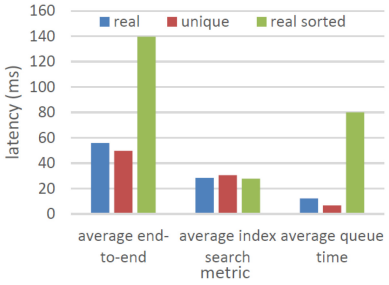

Fig. 24. Performance of various streams with different query temporal locality.

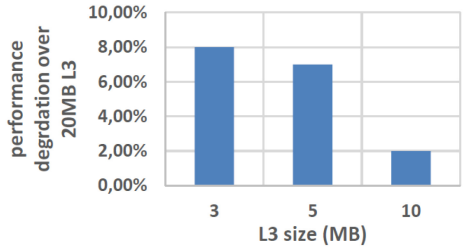

Fig. 25. LLC size exploration for single-core index search.

An interesting direction for future work is to explore the cost versus performance trade-off due to prefetching.

Next, we explore the sensitivity of index search to temporal locality. We compare three types of query streams: (a) a real query stream taken from the AOL log as it is, (b) a unique stream that has only unique queries (this is what we use in the rest of the article), and (c) the real query stream sorted alphabetically. One would expect that the sorted stream would provide the best performance and the unique stream the worst, due to good and bad query temporal locality, respectively. Figure 24 shows the (a) end-to-end average latency (the one perceived by the client), (b) the pure index search time that does not include queueing delay, and (c) the queueing time for the three different streams (using the 1I_8C@2.4GHz and 150 queries/sec traffic configuration). First, we examine the average index search times. The average index search times show that the unique query stream has slightly higher latency than the real stream and that the real stream latency is slightly higher than the sorted stream. This is what we expected, but because the differences are really small (unique index search time is 3ms higher than sorted), we can conclude that optimizing the query stream executed by an index server for temporal locality is not something that noticeably benefits the performance. Regarding average end-to-end latency, we observe that the sorted stream has much higher latency, which seems counter-intuitive. This high latency, however, is not due to the index search time but to the queueing time. Analysis, not shown in a graph, reveals that for the sorted stream, it is more likely to have high latency queries occurring back to back, which in turn increases the likelihood for larger queueing delay. Real stream also suffers from this effect; this is reflected by the fact that real stream has slightly higher queueing from the unique stream. In summary, we conclude that index search intrinsically has a cache-friendly behavior, and thus it is hard to gain large benefits from improving query temporal locality. Any significant differences in query latencies among the three different query streams are not attributed to the actual index search processing times but to queueing effects.

Last, the sensitivity of the index search to LLC size is explored. The low L3MPKI of the index search suggests that an index server could maintain its performance with a smaller L3 cache, which potentially can translate to either lower cost (cheaper CPU) or power savings (less leakage). In addition, it can translate to better data center utilization. For example, data center operators can collocate web search with other applications without hurting the QoS of web search. We use bubbles [26] to emulate a reduced cache size and analyze the index search degradation. Bubbles are processes that perform random accesses in a user-specified array. This way, they stress shared memory resources like LLC, DRAM, and so forth. We collocated one index server process with one bubble process. We change the bubble's array size to emulate various L3 sizes. Particularly,

we try bubble sizes of 17, 15, and 10 MB, which translate to emulation of 3, 5, and 10 MB L3 cache sizes (recall that our CPU has a 20MB LLC). The bubble is placed on a different core, and it is restricted to run only on that core. The index server is also restricted to use only one core, and the experiments are performed with 50 queries/sec traffic. Effectively, this way, one core index search performance with 3, 5, and 10 MB of L3 is emulated. Figure 25 shows the results of our evaluation. For a 10MB emulated cache, the index search shows negligible degradation. For a 5MB emulated cache, a 7% degradation is observed and for 3MB an 8% performance degradation. These results suggest than an acceptable LLC size would be slightly above 10MB. Below 10MB, the degradation is rather significant. Put another way, a core running our index search does not need all 20MB of LLC. The results imply that a single-core index search can safely collocate (will suffer negligible performance degradation) with at least one other process with a small memory footprint.

6 CONCLUSION

In this work, we thoroughly characterize using real hardware the Nutch search engine [8] that is based on the popular Lucene document search library. The main contributions of this work include (a) shedding light on the search engine's software architecture and functionality through application and micro-architectural characterization, (b) exploring the possibility of using low power servers for search engines and the effects of intra-server partitioning, and (c) studying the sources of index search performance degradation.

The key findings of our work are the following: (a) index search times scale linearly with the amount of the index dataset, (b) summary generation processing time does not scale with the amount of the dataset and remains constant, (c) index search is the most time consuming operation of the search engine, (d) we motivate the use of low power servers for index search, (e) we confirm that intra-server partitioning can help tail latencies with diminishing benefit with a higher rate of incoming traffic, (f) the majority of frequent index terms have balanced posting lists (at least for a small number of shards), (g) the use of index compression for storing the index adds to the amount of instructions executed per byte read and explains the high IPC and the good caching behavior of index search, (h) index search has small benefit from hardware prefetching and has low sensitivity to LLC size, and (i) idle states are detrimental to index search.

Our experimental setup characteristics, such as (a) posting list sorted by document ID, (b) absence of advanced document scoring methods, and (c) single-server analysis, are more similar to enterprise search engines rather than state-of-the-art large-scale web search engines. Nonetheless, our findings have implications for large-scale deployments. In general, for a fixed-size index and tail latency requirement, reducing the response time of a single server can help increase the amount of index a server can process and therefore reduce the number of servers that are needed to hold the entire index and used to service a query. Assuming that each server response time to a given query is a random variable that follows same response time distribution, the smaller the number of servers, the less probable it is to observe a worst-case latency. As part of our future work, we plan to explore, both analytically and empirically, how optimizations that improve the tail latency of a single server can help reduce the tail for systems with many servers. Moreover, for future work, we plan to explore dynamic index search parallelism, as well as dynamic frequency scaling based on query processing demands and the target QoS. We also plan to evaluate this benchmark using low power micro-servers.

ACKNOWLEDGMENTS

This work was partly supported by the Ministry of Education, Youth and Sports of the National Programme for Sustainability II (NPU II) under the project "IT4Innovations excellence in science–LQ1602". We thank the IT4Innovations Czech Republic National Supercomputing Center for

providing us with the infrastructure for conducting the experiments. In addition, we thank the anonymous reviewers for their constructive critique and feedback that helped improve the quality of the article.

REFERENCES

[1] Apache. 2012. Nutch Crawl Tutorial. Retrieved April 9, 2019 from https://wiki.apache.org/nutch/NutchTutorial.

[2] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, NY, 103–112.

[3] Claudine Badue, Ricardo Baeza-Yates, Berthier Ribeiro-Neto, and Nivio Ziviani. 2001. Distributed query processing using partitioned inverted files. In *Proceedings of the 8th Symposium on String Processing and Information Retrieval*. IEEE, Los Alamitos, CA, 0010.

[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 8, 3 (2013), 1–154.

[5] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro* 23, 2 (2003), 22–28.

[6] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and James Lin. 2012. Earlybird: Real-time search at Twitter. In *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE'12)*. IEEE, Los Alamitos, CA, 1360–1369.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, et al. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 205–220.

[8] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, et al. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *ACM SIGPLAN Notices* 47, 37–48.

[9] Zacharias Hadjilambrou, Marios Kleanthous, and Yiannakis Sazeides. 2015. Characterization and analysis of a web search benchmark. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'15)*. IEEE, Los Alamitos, CA, 328–337.

[10] Md E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 161–175.

[11] Damien Hardy, Marios Kleanthous, Isidoros Sideris, Ali G. Saidi, Emre Ozer, and Yiannakis Sazeides. 2013. An analytical framework for estimating TCO and exploring data center design space. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, Los Alamitos, CA, 54–63.

[12] Todd Hoff. 2009. Latency is everywhere and it costs you sales—How to crush it. High Scalability. Retrieved April 9, 2019 from http://www.highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[13] Urs Hölzle. 2010. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro* 30, 4 (2010), 1–2.

[14] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, et al. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, Los Alamitos, CA, 271–282.

[15] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2013. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, New York, NY, 155–168.

[16] S. Shunmuga Krishnan and Ramesh K. Sitaraman. 2013. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking* 21, 6 (2013), 2001–2014.

[17] J. Li, K. Agrawal, S. Elnikety, Y. He, I. Lee, C. Lu, K. S. McKinley, et al. 2016. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 14.

[18] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, New York, NY, 1–14.

[19] Linux. 2004. numatcl. Retrieved April 9, 2019 from http://linux.die.net/man/8/numactl.

[20] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News* 42, 301–312.

[21] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, et al. 2012. Scale-out processors. *ACM SIGARCH Computer Architecture News* 40, 500–511.

[22] Lucene. 2012. Lucene Scoring Explanation. Retrieved April 9, 209 from lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.
[23] Lucene. 2012. Lucene Variable Integer Format. Retrieved April 9, 2019 from https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/store/DataOutput.html#write VInt(int).
[24] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Vol. 1. Cambridge University Press.
[25] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in homogeneous warehouse-scale computers. *ACM SIGARCH Computer Architecture News* 41, 619–630.
[26] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 248–259.
[27] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. IEEE, Los Alamitos, CA, 319–330.
[28] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford InfoLab.
[29] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale'06)*, Vol. 152. Article 1.
[30] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web search using mobile cores. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*.
[31] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S. McKinley. 2013. Exploiting processor heterogeneity in interactive services. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. 45–58.
[32] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. 2011. Posting list intersection on multicore architectures. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, NY, 963–972.
[33] B. Vamanan, H. Bin Sohail, J. Hasan, and T. N. Vijaykumar. 2015. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, New York, NY, 585–597.
[34] Huafeng Xi, Jianfeng Zhan, Zhen Jia, Xuehai Hong, Lei Wang, Lixin Zhang, Ninghui Sun, et al. 2011. Characterization of real workloads of web search engines. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'11)*. IEEE, Los Alamitos, CA, 15–25.
[35] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News* 41, 607–618.

Received May 2018; revised March 2019; accepted March 2019